Serving Deep Neural Networks at the Cloud Edge for Vision Applications on Mobile Platforms

Zhou Fang, Dezhi Hong, Rajesh K. Gupta
Computer Science and Engineering, University of California, San Diego

ABSTRACT

The proliferation of high resolution cameras on embedded devices along with the growing maturity of deep neural networks (DNNs) has spawned powerful mobile vision applications. To enable applications on mobile devices, the offloading approach processes live video streams using DNNs on server-class GPU accelerators. However, their use in latency constrained applications is particularly challenging because of the large and unpredictable round-trip latency from mobile devices to the cloud computing resources. As a consequence, system designers routinely look for ways to offload to local servers at the cloud edge, known as the *cloudlet*. This paper explores the potential of serving multiple DNNs using the cloudlet model to implement complex vision applications on mobile devices. We present *DeepQuery*, a new mobile offloading system that is capable to serve DNNs with different structures for a wide range of tasks including object detection and tracking, scene graph detection, and video description. DeepQuery provides application programming interfaces to offload applications programed as Directed Acyclic Graphs of DNN queries, and employs data parallelization and input batching techniques to reduce processing delays. To improve GPU utilization, it co-locates real-time and delay-tolerant tasks on shared GPUs, and exploits a *predictive* and *plan-ahead* approach to alleviate resource contention caused by co-locating. We evaluate DeepQuery and demonstrate its effectiveness using several real world applications.

CCS CONCEPTS

• **Human-centered computing → Ubiquitous and mobile computing**; • **Computer systems organization → Cloud computing**; **Client-server architectures**;

KEYWORDS

mobile computing, cloud computing, edge computing, computer vision, deep neural networks

ACM Reference Format:
Zhou Fang, Dezhi Hong, Rajesh K. Gupta. 2019. Serving Deep Neural Networks at the Cloud Edge for Vision Applications on Mobile Platforms. In *Proceedings of 10th ACM Multimedia Systems Conference, Amherst, MA, USA, June 18–21, 2019 (MMSys '19),* 12 pages. https://doi.org/10.1145/3304109.3306221

1 INTRODUCTION

High resolution cameras are becoming the norm on mobile platforms such as smartphones, Google glasses [6], AWS DeepLens [5], etc. This enables a wide variety of vision applications that can extract contextual information from live video streams. For examples, intelligent personal assistants on smartphones answer queries in the form of voice and vision information to assist users [23]. Cognitive assistants on wearable glasses can also provide the environmental information to guide people with visual impairment [16]. Complementing these advances in hardware and platforms is the maturity of deep neural networks (DNNs) that are now widely adopted in computer vision applications. By quantitative measures, DNNs have achieved the state-of-the-art performance for the canonical classes of vision tasks at the object level (*e.g.*, recognition, detection, and tracking) [10, 25, 53]. We also start to see progress in tasks at higher semantic levels, such as human activity understanding [26], scene graph detection (object relationships) [33, 47], and video description [48]. The progresses in platforms and algorithms are at the threshold of a greatly enhanced ability of machines to understand video content to create intelligent applications.

To enable more accessible DNN models on mobile devices, many methods have been explored to attain low-power and real-time model inference, *e.g.*, designing lightweight models [28, 30, 45], pruning and quantizing models [17, 19, 46], and using hardware accelerators [8, 38, 50]. As an alternative to on-board computing, the mobile offloading approach augments embedded devices with resource-rich servers [11, 16, 18, 32] that are equipped with GPU [13, 22], FPGA [15], and/or ASIC [31] accelerators. These servers can be deployed either in local clusters, envisioned as cloudlet [41], or in the cloud [1–3]. Offloading DNN inference tasks to servers benefits from multiple computing resources in parallel, which is infeasible for on-board mobile computing.

MMSys '19, June 18–21, 2019, Amherst, MA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6297-9/19/06…$15.00
https://doi.org/10.1145/3304109.3306221

We envision emerging mobile video analytics applications that would employ multiple DNN models to accurately extract a rich class of information. For example, a vision-based smart hospitals [21] follows such a flow and deploys several DNNs for pedestrian detection and tracking as well as activity classification. The performance of video description can be boosted by combining 2D and 3D CNN features, and hand-crafted features from raw video streams and optical flows [49]. In doing so, the application involves several feature extraction models. In this paper, we motivate our work by the application in Figure 1 as an example — it extracts visual information at different semantic levels, including object, scene graph, and video description in natural language — which can be used to guide users or answer questions in cognitive assistant applications.

These applications would contain real-time (RT) workloads, such as object tracking in a live video [11], as well as non-RT delay-tolerant tasks such as describing a video clip [48]. Particularly, cognitive tasks would have stringent timing requirements and usually target at sub-second end-to-end delays to provide similar experiences to tasks conducted by humans [16]. For example, a previous study shows that on average it takes a human 370ms to recognize familiar faces and 620ms to recognize unfamiliar ones [39]. As a result, we would need a system that is capable of scheduling workloads with different timing requirements. A local cluster, *i.e.*, the cloudlet [41], holds promise for these RT workloads by providing timely offloading through low-latency high-bandwidth WiFi networks [11, 16]. It can reduce the network delays by hundreds of milliseconds compared to using remote cloud servers [12]. In addition, it avoids streaming large amounts of data over public networks to the cloud by aggregating and processing data on the edge servers [52].

To this end, we present *DeepQuery*, a new mobile offloading system that serves DNNs for live video analytics applications deployed on local servers at the network edge. DeepQuery provides new capabilities to simplify application deployment, reduce processing delay, and improve GPU resource utilization, through the following means:

Programming APIs: To support complex applications comprised of several tasks, DeepQuery provides a set of APIs to manage mobile data as key-value pairs on the servers, and employs queries to process data using DNNs. An application can thus be implemented as a Directed Acyclic Graph (DAG) of queries. In our design, intermediate data and applications states can be cached on servers to avoid redundant data transfer. For example, in Figure 1, object tracking and scene graph detecting tasks depend on the output of object detection. The DAG of queries contains such data dependency information, and the system automatically caches the detection output on servers for subsequent queries.

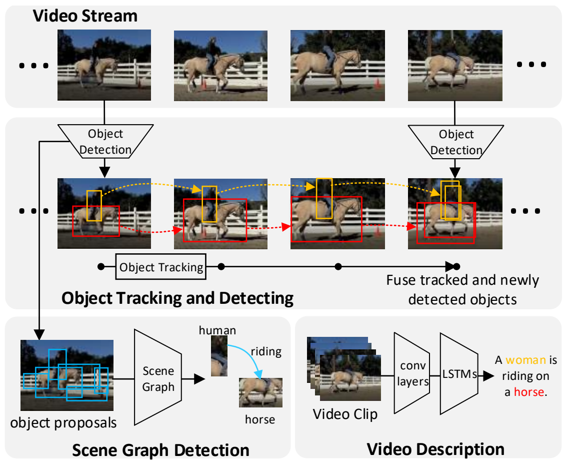


Figure 1: An application that retrieves visual information at different semantic levels (objects, scene graph, video description) using DNNs, from a live video stream. Our system provides APIs to program such complex applications, with infrastructure support to optimize DNN inference and scheduling methods to co-locate real-time (RT) and non-RT tasks on shared GPUs for high resource utilization.

Low-latency DNN Inference: DeepQuery accelerates DNN inference by providing the infrastructure for data parallelization. Consider, for instance, the Faster Region-based CNN (FRCNN) [40], a popular object detection model. It can be accelerated by classifying object proposals on multiple GPUs in parallel. FRCNN using Inception ResNet V2 feature extractor is 1.7 times faster on 4 GPUs. The speedup is 2.2 times faster for the more expensive FRCNN model using NasNet feature extractor, tested on NVIDIA GTX 1080 Ti GPUs.

High GPU Utilization: The GPU resource in a local cloudlet is limited compared to the cloud. It is thus valuable to co-locate real-time (RT) and non-RT workloads on shared servers for a higher resource utilization. Accordingly, GPU resources should be prioritized for RT workloads over non-RT workloads. Due to the lack of support for preemptive multi-tasking on GPUs, our proposed scheduler adopts a new *predictive* approach: It predicts future RT queries using application information provided by the clients, including DNN processing delays and network delays that are measured online. Dynamic batch size control is applied to non-RT workloads, in order to finish their processing before any future RT queries start to occupy the resource. Our scheduler also uses the predicted future GPU load for other purposes, *e.g.*, dynamic batch size selection in data parallelization, as well as worker selection for incoming queries for server load balance.

The primary contribution of this paper is the system design and implementation of DeepQuery. In addition to mobile videos, our system works for a broader class of video sources, *e.g.*, wired surveillance systems, in aspects of APIs, workload processing and scheduling, by handling network delays differently (Section 5.1). Additionally, our comprehensive evaluations include the characterization of several popular DNN models for various vision tasks, the mechanisms for offloading complex vision tasks efficiently, and the implementation of scheduling techniques to optimize co-located RT and non-RT DNN workloads for a high GPU resource utilization.

2 BACKGROUND AND RELATED WORKS

Mobile Offloading Systems: Recent works on mobile offloading have achieved substantial progress in optimizing end-to-end delay, mobile power consumption, and wireless network bandwidth usage. Gabriel [16] enables time-critical cognitive applications by deploying servers in a nearby cloudlet. The network delays are therefore minimized, since the server is only one wireless hop away from the mobile device. Recognizing the importance of DNN inferences on mobile platforms, MCDNN [18] studies the problem of scheduling variants of DNNs for performance-resource trade-off in an offloading setting. Neurosurgeon [32] proposes to partition DNN computations between the mobile and the cloud at the granularity of neural network layer to optimize processing delay and power consumption.

However, these systems only consider offloading tasks individually and ignore the data dependency between tasks. For complex vision applications consisting of multiple tasks with data dependency, such a simple offloading mechanism is inefficient due to the repetitive transfers of intermediate data between the client and server. By contrast, we propose fine grained data operations for mobile offloading — DNN models and other algorithms are programed as queries on data, which are highly flexible to program complex vision applications.

Video Analytics Systems: Video analytics systems that process a massive amount of queries on live video feeds in the cloud have been adopted in many application scenarios, *e.g.*, surveillance systems in cities or organizations [29, 36, 51] and smart hospitals [21]. Optasia [36] provides a set of SQL style APIs to build vision analytics applications, and further optimizes the queries to reduce the redundancy of computation. VideoStorm [51] programs an application as a DAG of transformations on data, which are executed by processing modules. The placement of processing modules and the configuration knobs of transforms are decided dynamically at runtime. VideoEdge [29] considers a hierarchy of resources with diverse computing and network capabilities, *e.g.*, camera devices, private clusters, and public clusters. Query optimization techniques are proposed to make trade-off between resource and accuracy.

These frameworks, however, are not designed for supporting and optimizing DNN workloads running on GPUs. In fact, these works consider stationary cameras for surveillance purposes as the primary use case. Compared to mobile cameras, many vision tasks can be simplified. For example, detection and tracking of moving objects can use the background subtraction method [14] running on processors. However, such lightweight methods fall short of analyzing videos from mobile cameras.

DNN Serving Systems: For a DNN model, inference is the process of making predictions through a forward network pass. A DNN serving system manages trained models and processes queries using the models. The DjiNN project [22] explores the idea of DNN-as-a-Service and designs a centralized DNN service infrastructure. It considers DNN models for image, speech, and natural language processing (NLP) applications. The system targets at warehouse-scale servers with high processing throughput as the primary goal. Clipper [13] is a low-latency serving system. It considers delay, throughput, and accuracy altogether, and adopts dynamic batching and model selection techniques to improve the service performance. These works consider object classification as the primary vision workload, which is in contrast to our envisioned scenario here: mobile vision applications piggybacked on local GPU servers using a variety of DNN models. We extend previous works by investigating a richer set of DNN models, and co-locating DNN inference tasks with different timing requirements on shared GPUs for a higher utilization.

3 DNN WORKLOAD

We first characterize the DNN models used in the example application (Figure 1) to motivate the design of DeepQuery.

3.1 Platform Specifications

The experiments to characterize the DNN models are conducted on servers running Ubuntu 16.04 with Intel Xeon Gold 6136 processors, installed with CUDA 9.0 and cuDNN 7.1. We use up to 4 NVIDIA GeForce GTX 1080 Ti GPUs (11GB memory and 11.3TFLOPs each device) on one server. The DNN models are implemented in Tensorflow 1.5.1 [7].

3.2 Object Detection in Videos

Video object detection (VID) detects and tracks multiple objects in a live video stream. It generates traces of objects in continuous frames, *i.e.*, tubelets. An object is represented by a bounding box with a class label and a unique object identifier. We adopt a generic VID scheme named tracking-by-detection [20]. It detects objects in static video frames, and then associates the detected object boxes across frames to obtain tubelets. To reduce the high computation overhead of object detection, a lightweight object tracking is used to track objects across frames until a new detection takes place. The Hungarian algorithm [34] is used to associate newly detected objects with currently tracked objects. It uses the

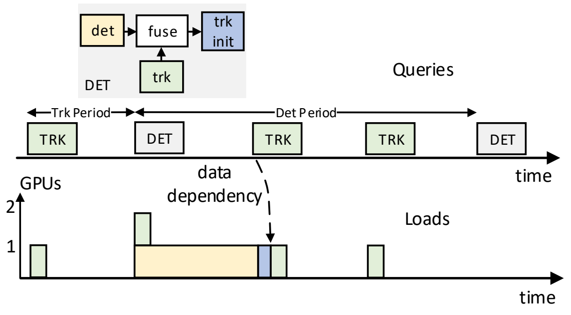


Figure 2: A video object detection application composed of object tracking (TRK) and detection (DET) tasks. A DET task contains sub-tasks including detecting objects in the new frame, tracking objects from the previous frame, fusing objects, and initializing the tracker. DeepQuery programs the sub-tasks as queries and uses predicted future GPU load in the scheduling algorithms to maximize server utilization.

intersection over union (IoU) of two object boxes as the association metric.

A VID application comprises TRK and DET tasks, as shown in Figure 2. A DET task consists of four sub-tasks: tracking objects from the previous frame, detecting objects in the current frame, fusing the results, and initializing the object tracker. The period is P_{TRK} for TRK and P_{DET} for DET. Figure 2 illustrates the GPU load that VID tasks generate. For a DET task, the tracking and detection sub-task can run in parallel on multiple GPUs, or multiple servers. If DET_i takes longer than P_{TRK} to complete, TRK_{i+1} will be delayed due to input data dependency, where i is the task index.

Object Detection: We use the Faster Region-based CNN [40] (FRCNN) model to detect objects. There are three stages: a Regional Proposal Network (RPN) stage detecting a fixed number of object proposals, a classification (CLS) stage classifying proposals, and a post-processing (POST) stage running the Non-Maximum Suppressing (NMS) algorithm to convert classified proposals to object bounding boxes.

The detection performance and speed of FRCNN detectors are decided by the feature extractor and the number of object proposals. Here we compare the delays of three detectors provided in the Tensorflow Detection Model Zoo [4]: Inception V2 (100 proposals), Inception ResNet V2 (300 proposals), and NasNet (300 proposals). Their detection performance (mAP) on the COCO dataset [35] is 28, 37, and 43, respectively [4]. Observed in Table 1, heavyweight models with higher detection accuracy are slower, with delays from hundreds of milliseconds to seconds, which limit their use in real-time applications. To speed up these object detectors, we present accelerating techniques that parallelize the CLS stage on multiple GPUs in Section 3.3.

Object Tracking: We use a multi-object tracker based on the Fully-Convolutional Siamese Networks (FC Siamese) [10].

Table 1: DNN inference delays (in ms) of RT tasks: The results are the average of 100 runs (with the standard deviation in parenthesis).

Object Detection					
Images	1	2	4	8	16
InceptionV2	38 (2)	64 (2)	125 (2)	239 (4)	478 (8)
IncResV2	370 (4)	728 (5)	1415 (8)	2863 (11)	-
NasNet	1025 (8)	2063 (8)	-	-	-
Object Tracking					
Objects	1	2	4	8	16
Trk-Init	6 (0)	10 (0)	18 (0)	37 (1)	74 (2)
Trk-Trk	19 (1)	32 (1)	60 (2)	115 (2)	230 (3)

Table 2: Precessing Delays of FRCNN stages using different numbers of GPUs (ms): Data parallelization can effectively accelerate DNN inference.

ResNet Inception V2				
#GPU	RPN	CLS	POST	Total
1	144 (3)	245 (1)	13 (2)	382 (3)
2	140 (5)	119 (1)	13 (2)	274 (7)
4	140 (3)	63 (3)	13 (3)	219 (8)
NasNet				
#GPU	RPN	CLS	POST	Total
1	244 (4)	747 (4)	13 (3)	1004 (8)
2	243 (3)	382 (2)	10 (1)	637 (3)
4	243 (3)	194 (1)	11 (1)	448 (3)

The tracker extracts CNN features from both the target and the search regions, and then localizes the target using a score map obtained from cross-correlation of the features. Accordingly, the tracking process contains two stages. The initialization (INIT) stage extracts features from a target and the tracking (TRK) stage extracts features from the search region in a new frame and localizes the target. To track multiple objects, the model takes multiple objects as the input of TRK, which has the form (im, [trk_states...]), where im is a video frame, trk_state is a tracking state including target feature and location. As an input batch for the Siamese Network, the input is transformed to ([search_regions...], [target_regions...]). The output are the bounding boxes of tracked objects and the tracking states with updated target feature and location.

Table 1 summarizes the delays for processing different numbers of targets as a batch. Using a batch of 8 images increases the throughput by a factor of 1.3 for INIT and TRK, compared with processing a single object. To improve the overall throughput, for tracking tasks which have few targets, several tasks can be batched as ($[\text{im}_i...]$, $[[\text{trk\_states}...]_i...]$). The batching should be aware of timing requirements for RT tasks, because it increases the processing delays.

3.3 Data Parallelization

We consider speeding up DNN inference using data parallelization on multiple GPUs in a server. For object detection, although many fast lightweight models have been proposed [28, 30], heavyweight DNNs [53] still demonstrate

Table 3: DNN inference delays of non-RT tasks (ms).

CNN Feature Extraction					
Images	1	2	4	8	16
ResNet-152	14 (1)	19 (1)	30 (2)	56 (2)	99 (2)
VGG-16	56 (1)	114 (3)	206 (4)	488 (5)	-
Scene Graph Detection					
Object Proposals	2	4	16	64	-
Scene Graph	62 (1)	59 (2)	71 (1)	249 (3)	-
Video Description					
Video clips	1	16	64	128	256
S2T	48 (4)	52 (2)	206 (14)	323 (14)	589 (24)

much superior performance [4]. The parallelization technique is able to achieve both high performance and low delay, which is valuable for RT applications. A FRCNN model can be expedited by processing the CLS stage on multiple GPUs in parallel. Each GPU classifies a portion of the proposals generated in the RPN stage. We test how much the delay can be reduced by evenly allocating the proposals on 1, 2 and 4 GPUs. As shown in Table 2, with parallelization using 4 GPUs, FRCNN (Inception ResNet V2) is 1.6 times faster, and the more compute intensive FRCNN (NasNet) model is 2.2 times faster.

We implement the infrastructure support for data parallelization in DeepQuery (see Section 4.4). It also works for other parallelizable models. For example, processing pipelines similar to FRCNN, which extract regional features from object proposals, have been adopted by a variety of models, *e.g.*, image description tasks using object features [9].

3.4 High-Level Vision Tasks

High-level vision tasks extract contextual information at higher semantic levels. In this work, we consider them as non-RT workloads and investigate scheduling techniques to co-locate them with RT tasks, as well as to reduce their influence on RT tasks.

CNN Feature Extraction: A feature extractor has the same architecture as a CNN based object classifier [24, 28, 53]. The difference is that a feature extractor removes the last fully-connected classification layer and outputs a feature vector instead of a classification label. Batching multiple queries in CNN inference is effective in improving throughput [13, 22]. We measure inference delay for two popular CNNs, VGG-16 [42] (130 million parameters) and ResNet-152 [24] (60 million parameters), as shown in Table 3.

Scene Graph Detection: A scene graph detection task (SG) takes object proposals in an image as input, detects objects and infers the relationships between objects. We consider the model proposed in [47] that has three stages: generating object proposals, extracting VGG-16 image features, and inferring objects as well as their relationships using iterative message passing. Different from object detection, object class

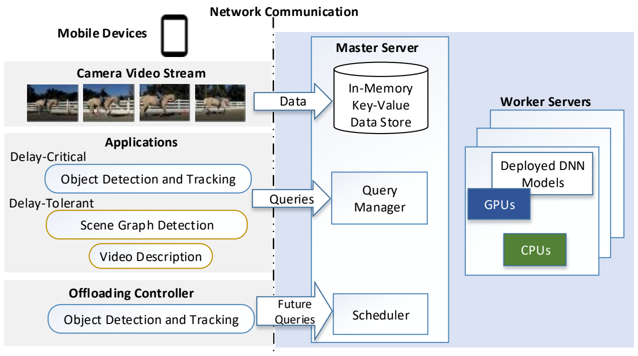


Figure 3: An overview of the DeepQuery system: The master server contains an in-memory database to cache the data, a query manager to manage queries that process data, and a scheduler to coordinate query executions. Queries are executed on the worker servers equipped with GPUs.

labels are inferred using both the features of objects and object relationships.

We measure SG delay excluding the object proposal stage, as shown in Table 3. The delay includes the VGG-16 delay in Table 3. Similar to multi-object tracking, the delay depends on image context, *i.e.*, the number of object proposals in this case. In our implementation, the proposals are generated from an object detector using a low threshold of detection confidence score (0.01).

Video Description: A video description task generates a sentence in natural language from a video clip. We adopt the sequence to sequence model [44] that takes the CNN features of video frames as input, which are encoded by a stacked LSTM with two hidden layers [27]. The LSTM decodes the features into a sequence of words. Our implementation takes video clips of 80 frames as input, with a CNN feature dimension of 4096. The serving system can improve the throughput of video description by batching video clips, as shown in Table 3.

4 DEEPQUERY SYSTEM

DeepQuery is a DNN model serving system for vision applications on mobile platforms. It is deployed on the cloudlet that consists of a cluster of GPU servers. Figure 3 illustrates the architecture of DeepQuery: Mobile clients offload vision applications using DeepQuery’s APIs to upload data and submit queries on the data. On the server side, the *master* server manages data in a distributed in-memory key-value database. When the input data for a query are complete, the master dispatches them to *worker* servers for processing, which deploy DNNs as micro-services. The *scheduler* selects workers to dispatch queries and controls the batch size of DNN input on each worker.

4.1 Application Timing Types

The system supports three types of DNN queries. **Streaming RT** queries are generated from continuous delay-critical mobile vision applications, which are considered as the primary workloads in many mobile offloading frameworks [11, 16]. The DeepQuery scheduler prioritizes the executions of this type of queries over the others. A second type are **Non-RT** queries, which are delay-tolerant. **Batching** queries are also non-RT and they process batches of input data. Non-RT queries do not have tight delay constraints and are served by the system in a best-effort manner. Mobile applications may offload a mixed set of RT and non-RT queries. In the example application in Figure 1, VID submits streaming RT TRK and DET queries, SG queries are non-RT, and CNN feature extraction of a video clip is a batching query.

4.2 Programming Model

A **query** defines a vision task to offload, using DNN models or other vision algorithms. An application can submit multiple queries to the server. The server creates a corresponding **job** for each query, which contains a DAG of processing **stages**. A stage is the atomic unit of data processing, corresponding to a DNN model or an algorithm served as a micro-service. For example, FRCNN can be split to a pipeline comprising RPN, CLS, and POST stages, where we can parallelize the CLS stage. The input and output data of stages are cached in memory (not the key-value database). A stage becomes executable when its preceding stages complete and all input data are ready. As long as a job has runnable stages, it stays in the ready queue and can be fetched by job runners. Therefore it can run in parallel on multiple GPUs.

4.3 APIs

DeepQuery APIs separate operations that manage data and queries that process data. There are three types of APIs:
Data Operations: Data operations manipulate data stored on the server, including imagery data from mobile cameras and processing results of queries. All data are stored in a distributed in-memory database as immutable key-value pairs. The basic operations are ADD, GET, DEL to write, read, and delete keyed data, respectively. The key contains a name field and an index field. For example, a client adds the ith frame to the database on server with "im:i" as the key, with "im" as the name and i as the index. The data store supports higher level stream and batch operations that wrap the basic operations to handle a collection of key-value data. For example, a video stream that contains frames with indices from i to j-1 can be represented as "im:[i:j]".
Queries: A query specifies the type of the algorithm or DNN model, the input and output data keys. The system routes the query to a worker server that serves the corresponding algorithm or model. After running the query, the output data are paired with the output keys. They are cached on

Algorithm 1: A Request of DET Queries in VID.

```
Request {
  dataOps=[
    DataOp(op=ADD, k="im:i", v=image),
    DataOp(op=GET, k="boxes:i") ],
  queries=[
    Query(model=TRK/TRK,
      inputs=["im:i", "trk_states:i-1"],
      outputs=["t_boxes:i", "trk_states:i"]),
    Query(model=DET,
      inputs=["im:i"],
      outputs=["d_boxes:i"]),
    Query(algorithm=FUSE_OBJECTS,
      inputs=["t_boxes:i", "d_boxes:i"],
      outputs=["boxes:i"]),
    Query(model=TRK/INIT,
      inputs=["im:i, boxes:i"],
      outputs=["trk_states:i"]) ]
  futureQueries=[
     FutureQuery(task=TRK/TRK,
       time=t_now+period, batchSize=nr_boxes)]
}
```

the server or sent back to the client, controlled by the data operations. A query may contain additional information such as the parameters of the vision algorithms used.
FutureQueries: The scheduler (see Section 5) of DeepQuery requires the information of future queries to predict GPU load. The prediction is based on measured DNN processing delays, network delays, and data dependencies between queries. The application client may adjust offloading configurations. For example, the VID application may adjust the period of tracking, or DNN model types, to tune the performance and computational overhead. The FutureQueries APIs are used to keep the scheduler updated with the future queries. When a streaming RT application sends the ith offloading request, it needs to attach the information about the $(i + 1)$ request next, including start times (t_{start}), DNN model types (m), and input batch sizes (b) of upcoming queries.

Algorithm 1 shows an example of offloading a DET task in VID. It contains four queries: tracking objects in a new frame ("im:i") using the tracking states ("trk_states:i-1"), detecting objects, fusing the tracking ("t_boxes:i"), and detecting ("d_boxes:i") the results. At the end, the tracking states are initialized for the fused object boxes. In the future queries field, the next is a TRK query, with a batch size estimated as the number of current tracked objects.

4.4 Worker Server and Query Processing

A worker server has one or more GPUs. A GPU is wrapped by the *JobRunner* module, which exposes APIs to deploy DNN micro-services and process queries, and can serve multiple DNN models. In the current implementation, we manually place models on GPUs using configuration files. Automatic model placement remains our future work. To use a model

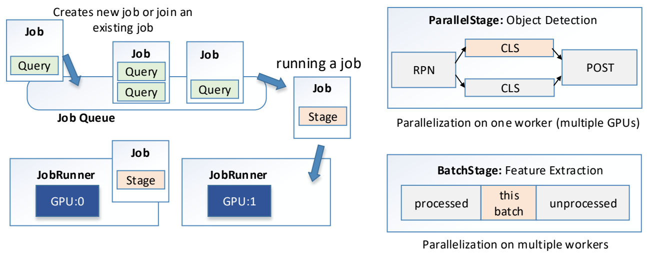

Creates new job or join an existing job
Job
Query
Job
Query
Query
Job
Query
Job Queue
running a job
Job
Stage
Job
Stage
JobRunner
GPU:0
JobRunner
GPU:1
ParallelStage: Object Detection
RPN
CLS
CLS
POST
Parallelization on one worker (multiple GPUs)
BatchStage: Feature Extraction
processed
this batch
unprocessed
Parallelization on multiple workers

Figure 4: Query processing on workers. A job is created for each query. A job runner manages job executions and loaded DNN models on a GPU. A ParallelStage can run on multiple GPUs to accelerate DNN inference using data parallelization. A BatchStage processes a batch of input data on multiple workers.

in DeepQuery, the micro-service interface must be implemented, including methods to initialize the model, batch multiple queries, and process input data.

The worker creates jobs to process queries. As shown in Figure 4, runnable jobs are buffered in a ready job queue, and each job runner fetches jobs from the queue to execute. A job runner executes one job at a time, which means the GPU device is exclusively used by a model. RT jobs are prioritized in the ready queue, and jobs with the same timing requirement are ordered by the query start timestamp t^{start}.

Parallel Stages: A *parallel* stage runs a DNN inference on multiple GPUs of one worker via data parallelization. It can speed up the parallelizalbe stages of DNN models, such as the CLS stage of FRCNN. The implementation adopts the *fork-join* model. It requires customized methods to split the input data and combine the output data. For FRCNN with 300 proposals, the outputs of the RPN stage include the coordinates of proposal boxes (shape 300×4, serialized size 4.8KB) and the extracted CNN features (shape $75 \times 100 \times 1088$, serialized size 32MB). A job runner can run a part of CLS by taking a sub-batch of the proposal boxes, and then combine the classifications from all runners to obtain the final result.

When we process a parallel stage with a batch size b on N_{gpu} GPUs, to dynamically decide the batch size for each GPU, we consider the running job on each GPU, when GPU G_k becomes free and tries to process a parallel stage, the optimal batch b_i to run on G_i satisfies

$$t_i^{end} + D_{b_i} = t_j^{end} + D_{b_j} \text{ and } \sum_{1}^{N_{gpu}} b_i = b, \tag{1}$$

where D_{b_i} is the estimated delay of processing b_i, and t_i^{end} is the time when G_i will become free. The methods to predict D_{b_i} and t_i^{end} will be presented in Section 5.1. Because the estimated D_{b_i} does not have a closed analytical form, to avoid expensive numerical methods, we use a linear function $D_b = kb$ to approximate it. The optimal b_i is then obtained as

$$b_i = [(D_{b_i} + \sum_{i=1}^{N_{gpu}} t_i^{end})/N_{gpu} - t_i^{end}]/D_{b_i} \cdot b, \tag{2}$$

The system obtains b_k using Equation (2) only for GPU G_k that is requesting a new job to run. It updates values $t_k^{end} = t_{now} + D_{b_k}$ and $b \leftarrow b - b_k$. Equation 2 is re-evaluated when another GPU tries to run this job.

Adaptive Batching: As discussed in the previous section, DNNs may take visual data in different forms as input, *e.g.*, images, objects, video clips. The worker automatically batches queries of lightweight DNN models in the ready job queue. Before creating a job for a query, the worker first checks whether there already exist this type of jobs in the ready queue. If so, the query is appended to an existing job. Considering delay increase due to batching, the system sets a default limit of the per job batch size for each DNN model. No more query can be added if the batch size of the job exceeds the threshold. A query can also specify a customized batch size limit, depending on its timing requirement.

4.5 Dispatching Queries to Workers

When the system has multiple workers, the scheduler routes a query to the appropriate worker based on its supported micro-service types and GPU load. It uses different strategies for different query types.

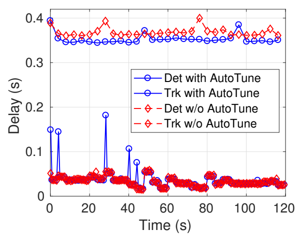
Streaming RT Queries: For a streaming RT query Q, the scheduler receives its information as a future query Q^{future}, attached to the previous offloading request. The scheduler selects the worker to dispatch Q at this time, ahead of when the real query arrives. It is necessary because Q^{future} must be allocated to one worker W_k, to update its set of future queries $\{Q^{future}\}_k$ for accurate GPU load prediction for each worker. The worker selection algorithms are based on predicted GPU contention and load (see Section 5.3).

Non-RT Queries: The worker to dispatch a non-RT query is selected when it arrives. Because non-RT tasks can only run when there are no runnable RT tasks. It selects the worker with the least total GPU load, calculated as $\sum_{q \in Q} D_q$, where D_q is the delay of query q, Q is the set of RT and non-RT queries allocated to a worker, including future RT queries.

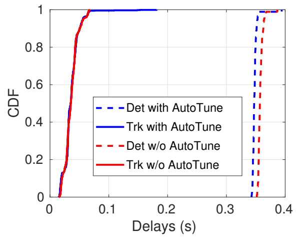
Batching Queries: A batching query can run on several workers in parallel. The master dispatches the query to a few workers and maintains the information of processed data in the batch. Before running the query, a worker communicates with the master to get a sub-batch of the input data. The sub-batch size is controlled by the scheduler to avoid affecting future RT queries on the worker (see Section 5.2).

5 SCHEDULING

The scheduler of the master server prevents non-RT queries from executing, or limits their input batch sizes, to avoid GPU resource contention with future RT queries. It also routes

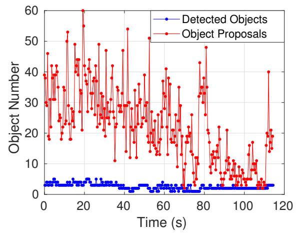


(a) DNN processing delays.

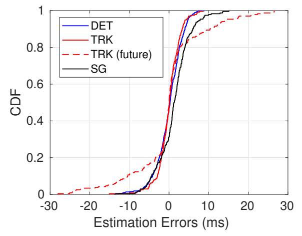


(b) CDFs of delays.

Figure 5: DNN processing delays for object tracking and detection with and without cuDNN AutoTune. When multiple models run on a shared GPU, AutoTune may incur delay spikes when it profiles the input data for a model. We switch off AutoTune to avoid spikes and improve the accuracy of delay predictions.



(a) Number of objects.



(b) Delay estimation errors.

Figure 6: (a) Number of object boxes in a sample video stream: the numbers of objects and proposals depend on the video context and may vary significantly. (b) Delay estimation performance: the varying data size causes errors in delay prediction for queries that take the data as an input batch, *e.g.*, TRK and SG queries.

queries to workers based on the estimated GPU load of each worker. These functionalities are based on the scheduler's capability to predict future GPU load.

5.1 Predicting GPU Workloads

For worker W_k, the scheduler predicts t_k^{end} and $\{Q^{future}\}_k$. t_k^{end} is the ending time of the current job. It is predicted as $t_k^{end} = t^{now} + D_b^m$ when the worker starts to run a job. D_b^m is the delay for running the DNN model m with a batch size b. $t_k^{end} = t^{now}$ for free workers.

The set $\{Q^{future}\}_k$ consists of future streaming RT queries allocated to W_k. It is predicted based on the starting times (t^{start}), batch sizes (b), and DNN model types (m) of future queries, provided by the FutureQueries APIs. The scheduler removes a future query Q_i^{future} from the set when it starts the corresponding query Q_i on W_k.

Interference on Shared GPU: Several DNN models can run on shared GPUs to improve resource utilization of cloudlet servers. A JobRunner executes one DNN inference job at a time. Intuitively, inferences do not exist when different DNNs run in sequence on a GPU. However, as shown in Figure 5a, delay spikes of tracking tasks (the blue lines) are observed when the tracking and detection models are on the same GPU. The *AutoTune* function that profiles input data to optimize batching processing in the *CuDNN* library is the source of delay spikes. Re-profiling happens when DNN workloads change and thus causes delay spikes. Figure 5a shows that the spikes are eliminated after turning off AutoTune. The advantages and drawbacks of AutoTune are clearly illustrated in Figure 5b. AutoTune speeds up FRCNN by around 10ms, whereas it results in long tails of TRK delays. In the system, we tune off AutoTune for machines that co-locate models for RT tasks and other workloads, in order to avoid unpredictable delay spikes.

Estimating DNN Delay: As discussed in Section 3, for DNN processing delays D_b^m, the variances are reasonably small comparing to the delays. Therefore D_b^m is regarded as a constant in the estimation. To capture the drift of D_b^m due to runtime fluctuation, the scheduler measures new samples D and updates the estimate as $D_b^m \leftarrow \alpha \cdot D_b^m + (1 - \alpha) \cdot D$. If the scheduler has no previous measurement of D_b^m, it selects two nearest batch sizes b_i and b_j whose delay measurements available and interpolates to estimate D_b^m. The estimate is replaced by the measured delay after processing the DNN.

The processing delay D_b^m of one image is varies for some tasks, *e.g.*, object tracking and scene graph detection for which b is the number of objects. Figure 6a shows the varying batch sizes for the two tasks, which are tracked objects (detection score over 0.5) and object proposals (score over 0.01). The estimation errors of tracking, detection (FRCNN Inception ResNet V2), and scene graph detection are given in Figure 6b. The errors are on the level of several milliseconds, which are reasonably low for load estimation.

In predicting future loads for streaming RT queries, the client needs to provide the batch sizes for future queries. For a task with varying batch sizes, it may introduce additional prediction errors. For example, for object tracking, we use $b_{i+1} = b_i$ assuming that the salient objects are slowly changing between two successive frames i and $i + 1$. The number of targets may change after running a DET task. It results in prediction errors of future query batch sizes. We give the errors of predicting D_b^m of TRK for b_{i+1} in Figure 6b, which are evidently larger than predicting D_b^m for b_i. These large errors only take place after DET queries, which have a portion of R_{TRK}/P_{DET} in all TRK queries. The errors result in performance degradation in predicting GPU loads and contentions for the scheduler.

Network Delay and Compensation: FutureQueries APIs provide starting times t_{client}^{start} of future queries on the client side. The time when a query arrives at the server t_{server}^{start}

is given by $t^{start}_{server} = t^{start}_{client} + d_{net} + \delta_{clock}$, where d_{net} is the upstream network delay, and δ_{clock} is the clock offset between the client and the server. We therefore estimate $\Delta^{start} = d_{net} + \delta_{clock}$ and obtain t^{start}_{server} accordingly for future queries. We measure samples Δ of Δ_{start} and use a smoothed value over the samples as the estimate $\Delta_{start} \leftarrow \beta \cdot \Delta_{start} + (1 - \beta) \cdot \Delta$. In this work, we consider a cloudlet setting using Wi-Fi networks in a university campus. Using Raspberry Pi 3 board as the mobile device, we measure the upstream delays offloading video frames with sizes from 21KB to 64KB. The 90 percentile is 23ms and the 99 percentile is 68ms.

5.2 Dynamic Batching for Non-RT Jobs

Resource utilization can be improved by co-locating RT and non-RT jobs on shared GPUs. It requires that non-RT jobs yield GPUs to any RT jobs that are ready to run. On CPUs, it is enabled by *preemptive multi-tasking*. However, without special handling in context switches enabled by driver extension, preemption incurs large overheads in both processing delay and throughput for GPU workload [37, 43]. Since such extensions are not yet generally available, this work solves the preemption problem by predicting future RT jobs and accordingly limiting the time budget for running non-RT jobs, in order to have them end before RT jobs start.

When a GPU (G_k) tries to run a non-RT job with a batch size b_k, the scheduler reduces b_k if the original value may affect a future RT query. It looks up the future RT queries on each GPU G_i ($\{Q^{future}_i\}$ with a size N_{query}) and obtains a sorted list of the start times of future queries t^{start}_i, where $i \in [1, N_{query}]$. The algorithm selects the maximal b_k that does not influence any future RT query. Given that there are currently N_{nrt} running non-RT jobs, the time budget for running them is limited by the latest N_{nrt} start times (t^{start}_i). The time budget for run a new non-RT job is given as $D_{budget} = t^{start}_k - t_{now}$ if $k \leq N_{query}$, where $k = N_{gpu} - N_{nrt}$. $D_{budget} = \infty$ if $k > N_{query}$. The scheduler then reduces the batch size b_k to satisfy $D_{b_k} \leq D_{budget}$, and then runs the non-RT job.

For a non-batching non-RT job, to avoid influencing RT queries, the scheduler estimates the time budget that a free GPU has to execute it. If the budget is not enough, instead of reducing the batch size, the scheduler keeps the job pending and the GPU idle.

5.3 Scaling to Multiple Workers

When there are multiple workers, the scheduler selects the best worker to dispatch each incoming query. As described in Section 4.5, for RT queries, when the server receives query Q_i with the information of the next query Q_{i+1}, the scheduler estimates the resource contention Φ_k after dispatching Q_{i+1} to worker W_k as follows:

$$\Phi = \int_{t_{now}}^{t_{end}} \phi(t)dt, \quad (3)$$

where $\phi(t)$ is a function of the number of requested GPUs $\theta(t)$ at time t, given by

$$\phi = \begin{cases} \theta(t) - N_{gpu}, & \text{if } \theta(t) > N_{gpu} \text{ or } (\theta(t) < N_{gpu} \text{ and } \Phi > 0). \\ 0, & \text{otherwise.} \end{cases} \quad (4)$$

Equation (3) and (4) accumulate the contention Φ when the system is overloaded, and consume Φ when it is underloaded. The scheduler selects the worker using $k = \text{argmax } \Phi_k$. When there is no GPU contention on any workers, we consider two strategies: (1) selecting the worker to which the previous query of the application is allocated (contention-affinity), to avoid transferring application state data (if any) between workers; and (2) selecting the worker with the least GPU load of running and future queries (named contention-load). We also implemented a baseline method that selects the worker with minimum load, without considering contention (load). The strategies are compared in Section 6.3.

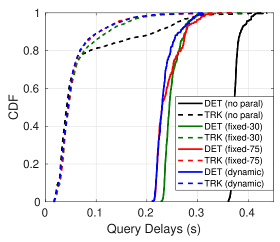
6 EVALUATION

We consider the scenarios that multiple clients run applications in Figure 1 on local servers as described in Section 3.1. We first evaluate the effect of data parallelization with dynamic batching (Section 6.1), and then demonstrate the effectiveness of co-locating RT and non-RT queries (Section 6.2), as well as the algorithms that dispatch queries to workers (Section 6.3).

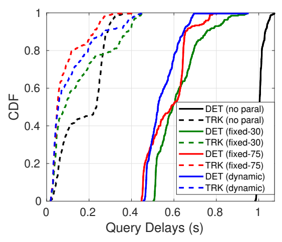
6.1 Dynamic Batching for Parallel Stages

To evaluate data parallelization and dynamic batching, we set 4 clients running the VID application on 4 GPUs, with two sets of configurations: P_{TRK} = 160ms, P_{DET} = 1.6s, and P_{TRK} = 200ms and P_{DET} = 2s. We test two FRCNN models, Inception ResNet V2 and NasNet, using 300 proposals, *i.e.*, $b = 300$. As described in Section 3.2, a DET task comprises of tracking, detection, and result fusion. We use the end times of result fusion to obtain DET delay. We use the query processing delay on servers as the performance metric. It includes server queueing and processing delays, but excludes network delays between clients and servers.

The comparisons include three baselines: (1) no parallelization; (2) a fixed batch size $b_i = 30$ and (3) $b_i = 75$. As shown in Figure 7a. We observe that parallelization effectively improves the delays for both TRK and DET queries. For both models, although a smaller batch size ($b_i = 30$) means finer grains of workload balance on multiple GPUs, it decreases the processing throughput. As a result, it has longer TRK and DET delays compared to $b_i = 75$. Dynamic batching control evidently improves the DET delay for both models.



(a) Inception ResNet V2.



(b) NasNet.

Figure 7: Distributions of DET and TRK query delays, using different batching strategies for the parallel CLS stage of FRCNN. It shows that using a small fixed batch reduces the throughput. However, a large fixed batch cannot be fully parallelized. The dynamic batching method selects the batch size considering all the running jobs on all GPUs to improve parallelism and reduce delay.

For NasNet, we observe that the TRK delay becomes larger than $b_i = 75$ when dynamic batching is used. This is because dynamic batching may allocate larger batches (> 75) in the CLS stage of FRCNN that compete GPU resource with TRK jobs. The result demonstrates that dynamic batching is effective in reducing delays for jobs with parallel stages.

6.2 Co-locating RT and Non-RT Queries

To evaluate the method for co-locating RT and non-RT queries, we process both types of queries on shared GPUs and measure the influence on the delays of RT queries.

Batching Queries: We evaluate the dynamic batching algorithm for non-RT jobs using a ResNet-152 feature extractor as an example. The algorithm should use as large batch sizes as possible to increase the throughput of ResNet-152, while avoiding affecting RT queries. A client runs a VID application (RT) on a GPU, with a low overhead configuration $P_{TRK} = 400$ms, $P_{DET} = 4$s. Another client submits a non-RT query extracting ResNet-152 features from a bath of 1024 images. The batching query may result in larger delays of VID queries. As the baselines, we use a fixed batch size 1, 4, 6, 8, and 16 to process the ResNet-152 query. As shown in Figure 8, a small batch size $b = 1$ results in a large processing delay of CNN, because of a low throughput. However, a large batch size $b = 16$ results in larger delays of TRK jobs, because it may occupy the GPU when a TRK is ready to run. The result shows that the dynamic batching methods can improve the throughput of non-RT jobs while maintaining low TRK delays. Although selecting an appropriate fixed batch size helps improve the performance as well, the value needs to be selected manually which is undesired.

The time slots of jobs on GPU is shown in Figure 9. The RT queries (DET, TRK) are plotted in red, and ResNet-152 is in yellow. It illustrates how a large batch size influences TRK jobs. Our algorithm selects the batch sizes dynamically to have non-RT jobs complete before future RT jobs start.

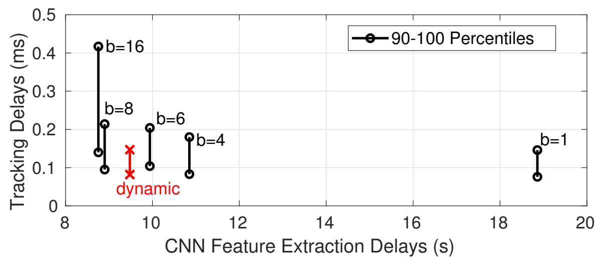


Figure 8: Delays of CNN (non-RT) and delay tail percentiles of TRK (RT) queries by different batching strategies. The dynamic batching method decides the maximal batch size that does not affect the RT jobs, which improves the delays of both RT and non-RT jobs.

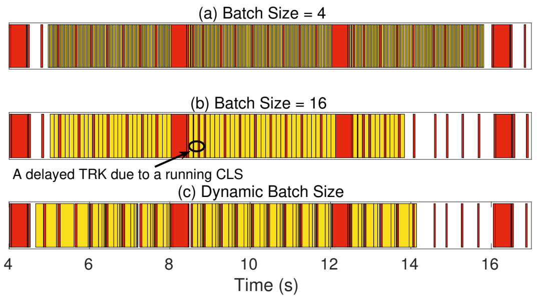


Figure 9: Time slots of DET, TRK (in yellow) and CNN jobs (in red) on one GPU. Using a small batch size for CNN (4) leads to a low processing throughput and a long delay, as shown by the total makespan of the yellow slots. A large batch size (16) results in significant contention with RT jobs: the long duration of the yellow slots affects the start times of the red slots. It illustrates how the dynamic batching method selects the batch size for non-RT jobs according to the time budget to complete before RT jobs.

Non-Batching Queries: For a non-RT query with a non-batching input, the scheduler runs it only when it does not affect future RT queries. We use scene graph queries processing 1 image as the example. Interestingly, we observe that the delay of SG depends on the number of object proposals. It is varying frame to frame in a video. The scheduler estimates the delay of a SG task to decide whether it can run.

In the experiments, a client running VID submits SG queries together with DET queries. SG takes the object proposals obtained from DET as input. The delays of SG queries are from server receiving the image to obtaining scene graphs, including the delays of DET. The experiments have 2 clients and one of them submits SG queries. As shown in Figure 10,

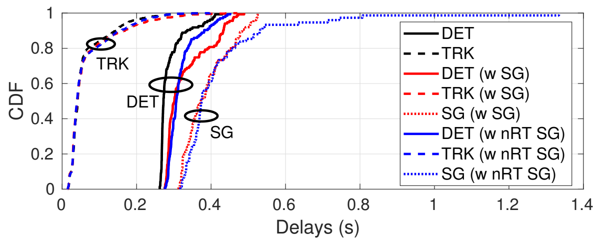


Figure 10: CDFs of DET, TRK and SG serving 2 clients, in the cases of no SG queries and one client submitting SG queries (with and without scheduling SG as non-RT jobs). It shows that when SG queries are non-RT, they yield resource to RT queries and are slower to complete. As a result, RT queries become faster as observed from the delay CDFs.

SG queries introduce additional delays to TRK and DET queries, which are originally 370ms (TRK) and 429ms (DET). When one client submits a SG query with each DET query, without scheduling SG as non-RT queries, TRK and DET delays increase to 471ms and 489ms. The scheduling method reduces the delays to 374ms (TRK) and 452ms (DET). In this case, because SG tasks can only run when there is enough time budget, their processing delays increase dramatically.

6.3 Worker Selection Strategies

To evaluate the algorithms to dispatch incoming queries to workers, Figure 11 compares the delays of DET and TRK queries when 4 clients run on a worker with 4 GPUs, and another setting with two workers, each with 2 GPUs. Object detection uses the Inception ResNet V2 FRCNN model. Because data parallelization causes difference in delays for workers with different numbers of GPUs, it is not used to do a fair comparison. The results show that the one worker setup has better performance, the tail delays of DET and TRK are 429ms and 437ms respectively. It is because there is no workload balance issues in this case. However, when distributing workloads on multiple workers, the imbalance of workloads may reduce the utilization of GPUs. Although the scheduler aims to balance GPU loads by minimizing contention, the prediction errors of network delays and DNN processing delays result in remaining contention. The result shows that the contention-load approach has the minimal tail DET and TRK delays, which are 597ms and 512ms. The CDFs of delays are similar for contention-affinity and load, but they have much larger tail query delays.

7 DISCUSSION

We next discuss the broader applications of our work beyond the constraints and assumptions of DeepQuery.

Combining DeepQuery with Conventional Serving Systems: Going beyond cloudlets, DeepQuery can also be backed

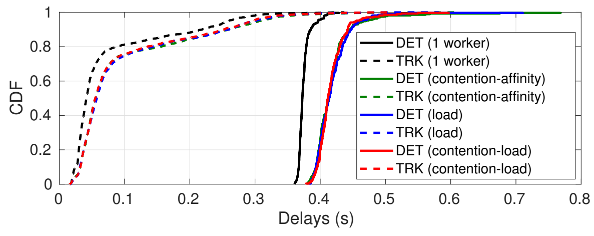


Figure 11: CDFs of DET and TRK query delays serving 4 clients, running on 1 worker with 4 GPUs, and 2 workers with 2 GPUs, using different scheduling strategies: contention-affinity, load, contention-load. It shows that delays increase when the computing resources are distributed onto two machines due to load imbalance. Contention-load performs the best in worker selection and achieves the smallest tail delays.

by cloud DNN serving systems [1–3]. In this *mobile-edge-cloud* deployment, DeepQuery processes RT queries on local servers, and a portion of delay-tolerant queries as well to fully utilize local compute resources. It can process delay-tolerant queries that have tight data dependency with RT queries, thus reducing the amount of data transferred to the cloud, while the remaining queries can be sent to the cloud for processing.

Tuning Mobile Offloading Online: Many mobile offloading frameworks adapt offloading configurations online to tune on-board energy consumption and application performance [11, 18]. For example, a VID application may adjust the detection period or the DNN model of detection, according to battery level, input video quality, and network condition. Although DeepQuery includes the predictability of mobile workloads, it supports such online changes because it only needs to predict for the next request. When offloading configuration is changed when it submits the request i, the changes will be applied starting from request i+1.

DNN Model Placement: There are several points to consider to place DNN models on GPUs: (1) the total memory overheads of the models on a GPU must be within the limit of the device and the system; (2) the number of GPUs to run a DNN model depends on the query rate of the model; (3) when co-locating different models together, we should consider their application timing requirements. For example, co-locating models for RT and non-RT queries helps improve resource utilization. Online model placement techniques need to be investigated in the future work.

8 CONCLUSION

In this paper, we present DeepQuery, a mobile offloading system that serves DNNs for computer vision applications. DeepQuery provides a new set of APIs to manage mobile data on servers as in-memory key-value pairs and program

DNN models as queries over the data. It provides the system support to accelerate DNN inference using data parallelization. To increase GPU resource utilization, it co-locates RT and non-RT tasks on shared GPUs. The resource contention on GPUs can be alleviated by controlling the batch sizes of non-RT tasks dynamically, based on the predictions of GPU resources requested by RT tasks. We demonstrate the effectiveness of DeepQuery through studying real world applications and a variety of DNN models.

REFERENCES

[1] AWS Machine Learning. https://aws.amazon.com/machine-learning/. [Online; accessed 16-July-2018].
[2] Azure Machine Learning. https://azure.microsoft.com/overview/machine-learning/. [Online; accessed 16-July-2018].
[3] Google Cloud Machine Learning. https://cloud.google.com/products/machine-learning/. [Online; accessed 16-July-2018].
[4] Tensorflow Detection Model Zoo. https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md. [Online; accessed 16-July-2018].
[5] The AWS DeepLens. https://aws.amazon.com/deeplens/. [Online; accessed 16-July-2018].
[6] The Google Glasses. https://x.company/glass/. [Online; accessed 16-July-2018].
[7] Martín Abadi et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.
[8] T. Ajayi et al. Celerity: An open source RISC-V tiered accelerator fabric. In *HOTCHIPS*, 2017.
[9] P. Anderson et al. Bottom-Up and Top-Down Attention for Image Captioning and Visual Question Answering. In *CVPR*, 2018.
[10] L. Bertinetto et al. Fully-Convolutional Siamese Networks for Object Tracking. In *ECCV Workshops*, 2016.
[11] Tiffany Yu-Han Chen et al. Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices. In *SenSys*, 2015.
[12] Zhuo Chen et al. An Empirical Study of Latency in an Emerging Class of Edge Computing Applications for Wearable Cognitive Assistance. In *SEC*, 2017.
[13] D. Crankshaw, , et al. Clipper: A Low-latency Online Prediction Serving System. In *NSDI*, 2017.
[14] A Elgammal et al. Non-parametric model for background subtraction. In David Vernon, editor, *ECCV*, pages 751–767, 2000.
[15] J. Fowers et al. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *ISCA*, 2018.
[16] K. Ha, , et al. Towards Wearable Cognitive Assistance. In *MobiSys*, 2014.
[17] S. Han et al. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv:1510.00149*, 2015.
[18] Seungyeop Han, , et al. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *MobiSys*, 2016.
[19] Song Han et al. EIE: efficient inference engine on compressed deep neural network. In *ISCA*, 2016.
[20] W. Han et al. Seq-NMS for Video Object Detection. *arXiv:1602.08465*, 2016.
[21] A. Haque et al. Towards Vision-Based Smart Hospitals: A System for Tracking and Monitoring Hand Hygiene Compliance. In *MLHC*, 2017.
[22] J. Hauswald, , et al. DjiNN and Tonic: DNN As a Service and Its Implications for Future Warehouse Scale Computers. In *ISCA*, 2015.
[23] J. Hauswald et al. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers. In *ASPLOS*, 2015.
[24] K. He et al. Deep Residual Learning for Image Recognition. In *CVPR*, 2016.
[25] K. He et al. Mask R-CNN. In *ICCV*, 2017.
[26] Fabian Caba Heilbron et al. ActivityNet: A large-scale video benchmark for human activity understanding. In *CVPR*, 2015.
[27] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.
[28] Andrew G. Howard et al. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv:1704.04861*, 2017.
[29] C.-C. Hung et al. Videoedge: Processing camera streams using hierarchical clusters. In *SEC*, 2018.
[30] F. N. Iandola et al. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv:1602.07360*, 2016.
[31] Norman P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, 2017.
[32] Y. Kang, , et al. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. In *ASPLOS*, 2017.
[33] R. Krishna et al. Visual Genome: Connecting Language and Vision Using Crowdsourced Dense Image Annotations. *Int. J. Comput. Vision*, 123(1):32–73, May 2017.
[34] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83—-97, 1955.
[35] T.-Y. Lin et al. Microsoft coco: Common objects in context. In *ECCV*, 2014.
[36] Y. Lu et al. Optasia: A relational platform for efficient large-scale video analytics. In *SoCC*, 2016.
[37] J. J. K. Park et al. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *ASPLOS*, 2015.
[38] J. Qiu et al. Going deeper with embedded fpga platform for convolutional neural network. In *FPGA*, 2016.
[39] Meike Ramon, Stephanie Caharel, and Bruno Rossion. The speed of recognition of personally familiar faces. *Perception*, 40(4):437–449, 2011.
[40] S. Ren et al. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.
[41] M. Satyanarayanan et al. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct 2009.
[42] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*, 2015.
[43] I. Tanasic et al. Enabling preemptive multiprogramming on GPUs. In *ISCA*, 2014.
[44] S. Venugopalan et al. Sequence to Sequence – Video to Text. In *ICCV*, 2015.
[45] B. Wu et al. Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In *CVPR*, 2017.
[46] J. Wu et al. Quantized convolutional neural networks for mobile devices. In *CVPR*, 2016.
[47] D. Xu, Y. Zhu, C. B. Choy, and L. Fei-Fei. Scene Graph Generation by Iterative Message Passing. In *CVPR*, 2017.
[48] J. Xu et al. MSR-VTT: A Large Video Description Dataset for Bridging Video and Language. In *CVPR*, 2016.
[49] T. Yao et al. MSR A MSM at ActivityNet challenge 2017. http://home.ustc.edu.cn/~panywei/paper/Activitynet17.pdf, 2017. [Online].
[50] C. Zhang et al. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *FPGA*, 2015.
[51] H. Zhang, , et al. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *NSDI*, 2017.
[52] T. Zhang et al. The Design and Implementation of a Wireless Video Surveillance System. In *MobiCom*, 2015.
[53] B. Zoph et al. Learning Transferable Architectures for Scalable Image Recognition. *arXiv:1707.07012*, 2017.